

Improved Algorithms for Computing k -Sink on Dynamic Path Networks

Binay Bhattacharya¹, Mordecai J. Golin², Yuya Higashikawa³,
Tsunehiko Kameda⁴, and Naoki Katoh⁵

1,4 School of Computing Science, Simon Fraser University, Vancouver, Canada
binay@sfu.ca, tikokameda@gmail.com

2 Dept. of Computer Science, Hong Kong Univ. of Science and Technology,
Hong Kong
golin@cse.ust.hk

3 Dept. of Information and System Engineering, Chuo University, Tokyo, Japan
higashikawa.874@g.chuo-u.ac.jp

5 School of Science and Technology, Kwansei Gakuin University, Hyogo, Japan
naoki.katoh@gmail.com

Abstract

We present a novel approach to finding the k -sink on dynamic path networks with general edge capacities. Our first algorithm runs in $O(n \log n + k^2 \log^4 n)$ time, where n is the number of vertices on the given path, and our second algorithm runs in $O(n \log^3 n)$ time. Together, they improve upon the previously most efficient $O(kn \log^2 n)$ time algorithm due to Arumugam et al. [1] for all values of k . In the case where all the edges have the same capacity, we again present two algorithms that run in $O(n + k^2 \log^2 n)$ time and $O(n \log n)$ time, respectively, and they together improve upon the previously best $O(kn)$ time algorithm due to Higashikawa et al. [10] for all values of k .

1998 ACM Subject Classification F.2.2

Keywords and phrases Facility location, k -sink, parametric search, dynamic path network

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

Investigation of evacuation problems dates back many years [7, 11]. The k -sink problem is to locate k sinks in such a way that every evacuee can evacuate to a sink as quickly as possible, when disasters, such as earthquakes and tsunamis, strike. The problem can be modeled by a network whose vertices represent the places where the evacuees are initially located and the edges represent possible evacuation routes. Associated with each edge is the transit time across it in either direction and its capacity in terms of the number of people who can enter it per unit time [7]. Madama et al. [12] solved this problem for the dynamic tree networks in $O(n \log^2 n)$ time under the condition that only a vertex can be a sink. For the 1-sink problem in the dynamic tree networks with uniform edge capacities, Higashikawa et al. proposed an $O(n \log n)$ algorithm [9] with the condition that the sink can be either at a vertex or on an edge.

On dynamic path networks with uniform edge capacities, it is straightforward to compute the 1-sink in linear time [2]. The k -sink problem for dynamic path networks with general and uniform edge capacities was solved in $O(kn \log^2 n)$ time by Arumugam et al. [1] and in $O(kn)$ time by Higashikawa et al. [10], respectively.



© Binay Bhattacharya, Mordecai J. Golin, Yuya Higashikawa, Tsunehiko Kameda and Naoki Katoh;
licensed under Creative Commons License CC-BY

Leibniz International Proceedings in Informatics

LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In this paper we present two algorithms for the k -sink problem for the dynamic path networks with general edge capacities. Together, they outperform all other known algorithms. We also present two algorithms for the dynamic path networks with uniform edge capacities. All our algorithms consists of two levels: feasibility tests at the lower level, and optimization at the higher level, making use of feasibility tests. Our results presented in this paper are the first algorithms that run in sub-quadratic time in n , regardless of the value of k , which can grow with n .

This paper is organized as follows. In the next section, we define our model and the terms that are used throughout the paper. In Sec. 3, we give an overview of our algorithm. Sec. 4 introduces a data structure called the critical cluster tree, which plays a central role in the rest of the paper. In Sec. 5, we identify two important tasks that form building blocks of our algorithms. and also discuss feasibility test. Sec. 6 presents several algorithms for uniform and general edge capacities. Finally, Sec. 7 concludes the paper.

2 Preliminaries

Let $P = (V, E)$ be a path network, whose vertices v_1, v_2, \dots, v_n are arranged from left to right in this order. For $i = 1, 2, \dots, n$, vertex v_i has an integral weight $w_i (> 0)$, representing the number of evacuees, and each edge $e_i = (v_i, v_{i+1})$ has a fixed non-negative length (distance) l_i and *capacity* c_i . We assume that all evacuees from a vertex evacuate to the same sink. We also assume that a sink has infinite capacity, so that the evacuees coming from the left and right of a sink do not interfere with each other. By $x \in P$, we mean that point x lies on either an edge or a vertex of P . For a vertex v , v^+ (resp. v^-) denotes the point just to the right (resp. left) of vertex v that is arbitrarily close to v . For $a, b \in P$, $a \prec b$ or $b \succ a$ means that a lies to the left of b . Let $d(a, b)$ denote the distance between a and b . If a and/or b lies on an edge, we use the prorated distance. The transit time for a unit distance is denoted by τ , so that it takes $d(a, b)\tau$ time to travel from a to b . Let $c(a, b)$ denote the minimum capacity of the edges on the subpath of P between a and b . Let $V[a, b]$ denote the set of vertices on the path from a to b . The subpath from a to b , including a and b , is denoted by $P[a, b]$. If a (resp. b) is excluded, we use $P(a, b]$ (resp. $P[a, b)$). Define

$$W[v_i, v_j] = \sum_{v_l \in V[v_i, v_j]} w_l. \quad (1)$$

Clearly $W[v_i, v_j]$ can be computed in constant time once we construct the array $\{W[v_1, v_j] \mid j = 1, 2, \dots, n\}$ in $O(n)$ time.

Given a subpath $P[v_i, v_j]$ of a dynamic path network P and a sink $s \in P[v_i, v_j]$, let $\Theta(s, [v_i, v_j])$ denote the evacuation time to s for the evacuees on $P[v_i, v_j]$. We also define the *L-cost* (resp. *R-cost*) of vertex $v_h \in V[v_i, v_j]$ in $P[v_i, v_j]$, as seen from $s \succ v_j$ (resp. $s \prec v_i$) to be the least evacuation time to s for all the evacuees on the vertices on $P[v_i, v_h]$ (resp. $P[v_h, v_j]$), assuming that they all arrive at s as continuously as possible. For any vertex $v_h \in V[v_i, v_j]$, its L-cost and R-cost are thus

$$\theta_L(s, [v_i, v_h]) = d(v_h, s)\tau + \frac{W[v_i, v_h]}{c(v_h, s)} \quad \text{for } s \succ v_j, \quad (2)$$

$$\theta_R(s, [v_h, v_j]) = d(s, v_h)\tau + \frac{W[v_h, v_j]}{c(s, v_h)} \quad \text{for } s \prec v_i. \quad (3)$$

Note that each of these functions is linear in the distance to s .

► **Lemma 1.** [1, 8] *Given a subpath $P[v_i, v_j]$ of a dynamic path network P and a sink $s \in P[v_i, v_j]$, $\Theta(s, [v_i, v_j])$ is represented by the following formula:*

$$\Theta(s, [v_i, v_j]) = \max \left\{ \max_{v_h \in V[v_i, s]} \theta_L(s, [v_i, v_h]), \max_{v_h \in V[s, v_j]} \theta_R(s, [v_h, v_j]) \right\}. \quad (4)$$

A problem instance is said to be (t, k) -feasible if exist k sinks such that every evacuee can reach a sink within time t . In our algorithms proposed in this paper, we perform preprocessing to construct a useful data structure, which makes (t, k) -feasibility test efficient.

3 Overall strategy of our algorithm

To carry out (t, k) -feasibility test, we repeatedly solve the following problems:

$L\text{-test}(P[v_i, v_j], t)$: It returns **yes** if every evacuee on a subpath $P[v_i, v_j]$ can reach a sink v_j^+ within time t . Otherwise it returns **no**.

$R\text{-test}(P[v_i, v_j], s, t)$: It returns **yes** if every evacuee on a subpath $P[v_i, v_j]$ can reach a sink s within time t where s is located on $(v_{i-1}, v_i]$. Otherwise it returns **no**.

As will be seen in Sec. 4, each of $L\text{-test}(P[v_i, v_j], t)$ and $R\text{-test}(P[v_i, v_j], s, t)$ can be done in $O(\log^2 n)$ time after constructing the data structure (called *critical cluster tree*). The critical cluster tree is a balanced binary search tree with height $O(\log n)$.

3.1 Feasibility test

The (t, k) -feasibility can be tested as follows: We first compute

$$l_1 = \max\{j \mid 1 \leq j \leq n, L\text{-test}(P[v_1, v_j], t) \text{ is "yes"}\}. \quad (5)$$

and find a sink $s_1 \in (v_{l_1}, v_{l_1+1}]$. We then compute

$$r_1 = \max\{j \mid l_1 + 1 \leq j \leq n, R\text{-test}(P[v_{l_1+1}, v_j], s_1, t) \text{ is "yes"}\}. \quad (6)$$

Repeating this procedure, if we eventually obtain $r_k = n$, (t, k) -feasibility test succeeds. If $r_k < n$, it fails. In fact, as will be seen in Sec. 5.2, (t, k) -feasibility test can be done in $O(k \log^3 n)$ time.

Using (t, k) -feasibility test as a subroutine, we can find a minimum value t^* such that (t^*, k) -feasibility test succeeds, which gives us an optimal evacuation time. This can be done by executing (t, k) -feasibility tests in binary search fashion.

3.2 Machineries in the data structure

The key idea is to use the balanced binary search tree \mathcal{T} with appropriate information stored at each node of the tree which enables us to execute each of $L\text{-test}(P[v_i, v_j], t)$ and $R\text{-test}(P[v_i, v_j], s, t)$ in $O(\log^2 n)$ time.

For $L\text{-test}(P[v_i, v_j], t)$, we need to compute

$$\Theta_L(v_i, v_j) = \max_{v_h \in V[v_i, v_j]} \theta_L(v_j^+, [v_i, v_h]). \quad (7)$$

Also for $R\text{-test}(P[v_i, v_j], s, t)$, we need to compute

$$\Theta_R(v_i, v_j, s) = \max_{v_h \in V[v_i, v_j]} \theta_R(s, [v_h, v_j]). \quad (8)$$

$L\text{-test}(P[v_i, v_j], t)$ succeeds if and only if $\Theta_L(v_i, v_j) \leq t$ and $R\text{-test}(P[v_i, v_j], s, t)$ succeeds if and only if $\Theta_R(v_i, v_j, s) \leq t$.

For leaf nodes $l(v_i)$ and $l(v_j)$ in \mathcal{T} which correspond to v_i and v_j , respectively, let u be the least common ancestor of \mathcal{T} . Then in the subtree $\mathcal{T}(u)$ with the root u in \mathcal{T} , we can identify a set of vertex-disjoint subpaths which covers vertices of $P[v_i, v_j]$ such that the number of such subpaths is $O(\log n)$, and every subpath corresponds to the set of leaves that a subtree $\mathcal{T}(u')$ spans for some node u' in $\mathcal{T}(u)$. Let $\mathcal{P}[v_i, v_j]$ denote the set of such subpaths.

In the rest of this section, we only show how to compute $\Theta_L(v_i, v_j)$ since $\Theta_R(v_i, v_j, s)$ is symmetric, so can be similarly computed. The computation of (7) reduces to

$$\Theta_L(v_i, v_j) = \max_{P[v_l, v_r] \in \mathcal{P}[v_i, v_j]} \left\{ \max_{v_h \in V[v_l, v_r]} \theta_L(v_j^+, [v_i, v_h]) \right\}. \quad (9)$$

To evaluate (9), we need to compute

$$\max_{v_h \in V[v_l, v_r]} \theta_L(v_j^+, [v_i, v_h]) = \max_{v_h \in V[v_l, v_r]} \left\{ d(v_h, v_j^+) \tau + \frac{W[v_i, v_h]}{c(v_h, v_{j+1})} \right\} \quad (10)$$

for every subpath $P[v_l, v_r] \in \mathcal{P}[v_i, v_j]$. Since $v_i \leq v_l \leq v_r \leq v_j$, the right side of (10) is rewritten as

$$\max_{v_h \in V[v_l, v_r]} \left\{ d(v_h, v_r) \tau + d(v_r, v_j^+) \tau + \frac{W[v_i, v_{l-1}] + W[v_l, v_h]}{\min\{c(v_h, v_r), c(v_r, v_{j+1})\}} \right\}. \quad (11)$$

Suppose that u' is a node of \mathcal{T} spanning $P[v_l, v_r]$. Then, to facilitate the computation of (11) for general case, we will prepare at node u' the following machinery that allows us to compute

$$\text{cost}_{u'}^L(W, C) = \max_{v_h \in V[v_l, v_r]} \left\{ d(v_h, v_r) \tau + \frac{W + W[v_l, v_h]}{\min\{c(v_h, v_r), C\}} \right\} \quad (12)$$

in $O(\log n)$ time once W and C are given. Here W and C are unknown parameters. This part will be explained in more detail in Sec. 4.

4 Data structures for the edge-capacitated case

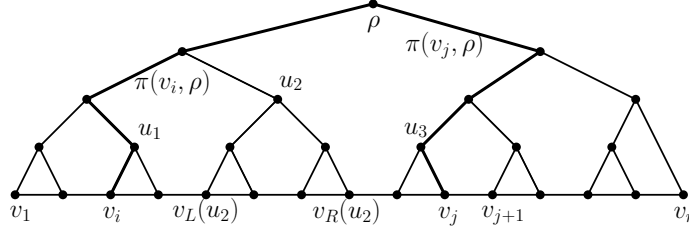
We want to perform (t, k) -feasibility tests for many different values of completion time t . Therefore, it will be useful to spend some time during preprocessing to construct data structures which facilitate those tests. Let us consider an arbitrary subpath $P[v_i, v_j]$, where $i \leq j$. The vertex $v_k \in V[v_i, v_j]$ that maximizes $\theta_L(x, [v_i, v_k])$ at $x \succ v_j$ (resp. $\theta_R(v_{i-1}, [v_k, v_j])$ at $x \prec v_i$) is called the *L-critical vertex* (resp. *R-critical vertex*) of $P[v_i, v_j]$ w.r.t. x , and the corresponding subpath $P[v_i, v_k]$ (resp. $P[v_k, v_j]$) is called the *L-critical cluster* (resp. *R-critical cluster*) of $P[v_i, v_j]$ w.r.t. x . It is easy to show the following proposition.

► **Proposition 1.** The L-critical (resp. R-critical) vertex/cluster w.r.t. x is the same for all points x on an edge, excluding its left (resp. right) end vertex. ◀

Therefore, we can talk about a critical vertex/cluster w.r.t. an edge. The L-critical vertex of $P[v_i, v_j]$ w.r.t. edge (v_j, v_{j+1}) (resp. R-critical vertex of $P[v_i, v_j]$ w.r.t. edge (v_{i-1}, v_i)) is denoted by $c_L^{[i, j]}$ (resp. $c_R^{[i, j]}$), and L-critical cluster of $P[v_i, v_j]$ w.r.t. edge (v_j, v_{j+1}) (resp. R-critical cluster of $P[v_i, v_j]$ w.r.t. edge (v_{i-1}, v_i)) is denoted by $C_L^{[i, j]}$ (resp. $C_R^{[i, j]}$). We thus have $C_L^{[i, j]} = P[v_i, c_L^{[i, j]}]$ and $C_R^{[i, j]} = P[c_R^{[i, j]}, v_j]$.

4.1 Critical cluster tree

We first construct the *critical cluster tree* (or *CC-tree* for short), \mathcal{T} , with root ρ , whose leaves are the vertices of P , arranged from left to right. It is a balanced tree with height $O(\log n)$. In balancing, the vertex weights are not considered. See Fig. 1, where $\pi(v_i, \rho)$ denotes the path from v_i to root ρ . For a node¹ u of \mathcal{T} , let $\mathcal{T}(u)$ denote the subtree rooted



■ **Figure 1** CC-tree \mathcal{T} .

at u , let u_l (resp. u_r) be its left (resp. right) child node, and let $v_L(u)$ (resp. $v_R(u)$) denote the leftmost (resp. rightmost) vertex on P that belongs to $\mathcal{T}(u)$. We say that $\mathcal{T}(u)$ *spans* subpath $P[v_L(u), v_R(u)]$ and also u spans $P[v_L(u), v_R(u)]$. At node u , we store two sorted lists of capacities $c(v_h, v_R(u))$ for $v_h \in V[v_L(u), v_R(u)] \setminus \{v_R(u)\}$ and $c(v_L(u), v_h)$ for $v_h \in V[v_L(u), v_R(u)] \setminus \{v_L(u)\}$. The list of $c(v_h, v_R(u))$ can be computed in the decreasing order in h in $O(|\mathcal{T}(u)|)$ time. Symmetrically, the list of $c(v_L(u), v_h)$ can be also computed in $O(|\mathcal{T}(u)|)$ time.

For $L\text{-test}(P[v_i, v_j], t)$ and $R\text{-test}(P[v_i, v_j], s, t)$ mentioned in Sec. 3, we need to determine $c_L^{[i,j]}$ and $c_R^{[i,j]}$, respectively. To do this, for every highest node u of \mathcal{T} spanning a subpath of $P[v_i, v_j]$ we will prepare a machinery at each node u of \mathcal{T} that allows us to compute

$$\text{cost}_u^L(W, C) = \max_{v_h \in V[v_L(u), v_R(u)]} \left\{ d(v_h, v_R(u))\tau + \frac{W + W[v_L(u), v_h]}{\min\{c(v_h, v_R(u)), C\}} \right\}, \quad (13)$$

$$\text{cost}_u^R(W, C) = \max_{v_h \in V[v_L(u), v_R(u)]} \left\{ d(v_L(u), v_h)\tau + \frac{W[v_h, v_R(u)] + W}{\min\{C, c(v_L(u), v_h)\}} \right\}. \quad (14)$$

in $O(\log n)$ for arbitrary W and C .

Suppose that we have such a machinery at each node u . Once $P[v_i, v_j]$ is given, for a node u spanning a subpath of $P[v_i, v_j]$, W and C are given as $W[v_i, v_{l-1}]$ (where $v_l = v_L(u)$) and $c(v_R(u), v_j)$ in (13), respectively. Let us call a vertex which achieves the maximum value in (13) w.r.t. u , $W = W[v_i, v_{l-1}]$ and $C = c(v_R(u), v_j)$ an *L-critical candidate* of $P[v_i, v_j]$. Then one of L-critical candidates of $P[v_i, v_j]$ must be $c_L^{[i,j]}$, which implies that we can do an $L\text{-test}(P[v_i, v_j], t)$ in $O(\log^2 n)$ time since there are $O(\log n)$ nodes spanning vertex-disjoint subpaths of $P[v_i, v_j]$. Similarly, an *R-critical candidate* of $P[v_i, v_j]$ is defined, and one of R-critical candidates of $P[v_i, v_j]$ is $c_R^{[i,j]}$, thus $R\text{-test}(P[v_i, v_j], s, t)$ can be done in $O(\log^2 n)$ time.

To compute $\text{cost}_u^L(W, C)$, one idea is to prepare a two-dimensional table at node u which returns the L-critical vertex of $P[v_i, v_j]$ for queries of W and C , but it takes much time and space in total. Instead of this, we actually store at node u two linear tables of vertices in

¹ We use the term “node” here to distinguish it from the vertices on the path. A vertex, being a leaf of \mathcal{T} , is considered a node, but an interior node of \mathcal{T} is not a vertex.

$V[v_L(u), v_R(u)]$: one returns a vertex $v_u^1(W)$ for a query of W which achieves

$$\max_{v_h \in V[v_L(u), v_R(u)]} \left\{ d(v_h, v_R(u))\tau + \frac{W + W[v_L(u), v_h]}{c(v_h, v_R(u))} \right\}, \quad (15)$$

and the other one returns a vertex $v_u^2(C)$ for a query of C which achieves

$$\max_{v_h \in V[v_L(u), v_R(u)]} \left\{ d(v_h, v_R(u))\tau + \frac{W[v_L(u), v_h]}{C} \right\}. \quad (16)$$

We call the first table *the left weight table* of u and the other *the left capacity table* of u . Also, to compute $\text{cost}_u^R(W, C)$, we store similar two tables at node u , called *the right weight table* of u and *the right capacity table* of u . Note that for each leaf node $u = v_i$ (which is a vertex of P), tables always return v_i itself for any W and C . In Sec. 5.1, we will show how to use these tables to compute (13) and (14).

4.2 CC-tree construction

In this section, we show how to construct the left weight table and the left capacity table at a node u of \mathcal{T} . Note that in the construction of CC-tree, we can construct tables of u without using any information stored at children u_l and u_r .

(a) **Weight table:** For a vertex $v_h \in V[v_L(u), v_R(u)]$, let $f_h^1(W)$ denote a function of W such that

$$f_h^1(W) = \alpha_h^1 W + \beta_h^1, \quad (17)$$

where $\alpha_h^1 = 1/c(v_h, v_R(u))$ and $\beta_h^1 = d(v_h, v_R(u))\tau + W[v_L(u), v_h]/c(v_h, v_R(u))$. Then the equation (15) can be rewritten as

$$\max_{v_h \in V[v_L(u), v_R(u)]} f_h^1(W). \quad (18)$$

Note that once we compute the upper envelope of $f_h^1(W)$ for all $v_h \in V[v_L(u), v_R(u)]$, it can return a vertex $v_u^1(W)$ for a query of W which achieves (18), which is equivalent to the left weight table of u . Here $f_h^1(W)$ is a linear function in W and α_h^1 is decreasing in h . Using the concept of duality of lines and points in 2-D, it is known that computing the upper envelope of lines is equivalent to computing the lower convex hull of points [3, 14]. As noted in [14], it is known that if points are sorted in x -coordinates, the convex hull can be computed in linear time by using the Graham scan algorithm [6]. Summarizing these facts, we can obtain the left weight table of u in $O(|\mathcal{T}(u)|)$ time,

(b) **Capacity table:** For a vertex $v_h \in V[v_L(u), v_R(u)]$, let $f_h^2(1/C)$ denote a function of $1/C$ such that

$$f_h^2(1/C) = \alpha_h^2 \cdot (1/C) + \beta_h^2, \quad (19)$$

where $\alpha_h^2 = W[v_L(u), v_h]$ and $\beta_h^2 = d(v_h, v_R(u))\tau$. Then the equation (16) can be rewritten as

$$\max_{v_h \in V[v_L(u), v_R(u)]} f_h^2(1/C). \quad (20)$$

Here $f_h^2(1/C)$ is a linear function in $1/C$ and α_h^2 is increasing in h , we thus can compute the upper envelope of $f_h^2(1/C)$ for all $v_h \in V[v_L(u), v_R(u)]$ in $O(|\mathcal{T}(u)|)$ time, which is equivalent to the left capacity table of u (similarly to (a)).

► **Lemma 2.** *Given a dynamic path network with n vertices and general edge capacities, we can construct its CC-tree, \mathcal{T} , in $O(n \log n)$ time.*

Proof. We construct two sorted lists of capacities, two weight tables and two capacity tables at every node u one by one (which does not need to be performed bottom up). As mentioned above, these all can be constructed in $O(|\mathcal{T}(u)|)$ time. For a non-negative integer d , let $U(d)$ denote a set of nodes of \mathcal{T} such that each node $u \in U(d)$ is located at depth d from root ρ (see Fig. 1). Therefore, letting h be the height of \mathcal{T} , the total time required to construct \mathcal{T} can be represented as $\sum_{d=0}^h \sum_{u \in U(d)} O(|\mathcal{T}(u)|)$. We here have $h = O(\log n)$ and $\sum_{u \in U(d)} O(|\mathcal{T}(u)|) = O(n)$ since for a fixed d , $\mathcal{T}(u)$ for all $u \in U(d)$ are vertex-disjoint, thus the total time is $O(n \log n)$. ◀

5 Two main tasks

There are two useful tasks that we can call upon repeatedly. **Task 1** is to find a maximal subpath $P[v_a, v_d]$, given the starting vertex v_a , such that we can place a 1-sink on it to enable all the evacuees to evacuate to it within time t . **Task 2** is to find a 1-sink on a given subpath $P[v_i, v_j]$. We want to construct an algorithm for each of these tasks.

In the two algorithms, given a subpath $P[v_i, v_j]$ and a node u of \mathcal{T} spanning a subpath $P[v_l, v_r]$ (i.e., $v_l = v_L(u)$ and $v_r = v_R(u)$) of $P[v_i, v_j]$, we need to compute (13) with $W = W[v_i, v_{l-1}]$ and $C = c(v_r, v_j)$, and (14) with $W = W[v_{r+1}, v_j]$ and $C = c(v_i, v_l)$. We first show the following lemma.

► **Lemma 3.** *Assume that the CC-tree, \mathcal{T} , is available. Then, given a subpath $P[v_i, v_j]$ and a node u of \mathcal{T} spanning a subpath $P[v_l, v_r]$ of $P[v_i, v_j]$, the L-critical candidate and the R-critical candidate of $P[v_i, v_j]$ belonging to $P[v_l, v_r]$ can be computed in $O(\log n)$ time.*

Proof. We only prove the case of the L-critical candidate, letting $W = W[v_i, v_{l-1}]$ and $C = c(v_r, v_j)$ (the proof for the R-critical candidate is symmetric).

Suppose that there exists an integer h^* satisfying $l \leq h^* \leq r - 2$ such that $c(v_{h^*}, v_r) \leq C$ and $c(v_{h^*+1}, v_r) > C$ (if does not exist, $c(v_h, v_r) \leq C$ for every h satisfying $l \leq h \leq r - 1$ or $c(v_h, v_r) > C$ for every h satisfying $l \leq h \leq r - 1$). Note that such h^* uniquely exists since $c(v_h, v_r)$ is increasing in h . We first separate $P[v_l, v_r]$ to two subpaths $P_1 = P[v_l, v_{h^*}]$ and $P_2 = P[v_{h^*+1}, v_r]$, which can be done in $O(\log n)$ time by binary search over the sorted list of capacities stored at u . Letting $V_1 = V[v_l, v_{h^*}]$ and $V_2 = V[v_{h^*+1}, v_r]$, we then consider

$$\begin{aligned} \text{cost}_u^L(W, C, P_1) &= \max_{v_h \in V_1} \left\{ d(v_h, v_r)\tau + \frac{W + W[v_l, v_h]}{\min\{c(v_h, v_r), C\}} \right\} \\ &= \max_{v_h \in V_1} \left\{ d(v_h, v_r)\tau + \frac{W + W[v_l, v_h]}{c(v_h, v_r)} \right\}, \end{aligned} \quad (21)$$

and

$$\begin{aligned} \text{cost}_u^L(W, C, P_2) &= \max_{v_h \in V_2} \left\{ d(v_h, v_r)\tau + \frac{W + W[v_l, v_h]}{\min\{c(v_h, v_r), C\}} \right\} \\ &= \max_{v_h \in V_2} \left\{ d(v_h, v_r)\tau + \frac{W + W[v_l, v_h]}{C} \right\} \\ &= \max_{v_h \in V_2} \left\{ d(v_h, v_r)\tau + \frac{W[v_l, v_h]}{C} \right\} + \frac{W}{C}. \end{aligned} \quad (22)$$

Note that $\text{cost}_u^L(W, C) = \max\{\text{cost}_u^L(W, C, P_1), \text{cost}_u^L(W, C, P_2)\}$. By binary search over the left weight table of u , we can identify a vertex v_1^* maximizing $\{d(v_h, v_r)\tau + (W +$

$W[v_l, v_h])/c(v_h, v_r)\}$ for $v_h \in V_1 \cup V_2$ in $O(\log n)$ time. Similarly, using the left capacity table of u , we can identify a vertex v_2^* maximizing $\{d(v_h, v_r)\tau + W[v_l, v_h]/C\}$ for $v_h \in V_1 \cup V_2$ in $O(\log n)$ time. Note that if $v_1^* \in V_2$,

$$\begin{aligned} \text{cost}_u^L(W, C, P_1) &\leq d(v_1^*, v_r)\tau + \frac{W + W[v_l, v_h]}{c(v_1^*, v_r)} \\ &< d(v_1^*, v_r)\tau + \frac{W + W[v_l, v_h]}{C} \leq \text{cost}_u^L(W, C, P_2). \end{aligned}$$

and if $v_2^* \in V_1$,

$$\begin{aligned} \text{cost}_u^L(W, C, P_2) &\leq d(v_2^*, v_r)\tau + \frac{W + W[v_l, v_h]}{C} \\ &\leq d(v_1^*, v_r)\tau + \frac{W + W[v_l, v_h]}{c(v_2^*, v_r)} \leq \text{cost}_u^L(W, C, P_1), \end{aligned}$$

which implies that $v_1^* \in V_2$ and $v_2^* \in V_1$ never occur simultaneously. Therefore, if $v_1^* \in V_2$, $v_2^* \in V_2$ and $\text{cost}_u^L(W, C, P_1) < \text{cost}_u^L(W, C, P_2)$, thus v_2^* is the L-critical candidate of $P[v_i, v_j]$. If $v_2^* \in V_1$, $\text{cost}_u^L(W, C, P_1) \geq \text{cost}_u^L(W, C, P_2)$ and $v_1^* \in V_1$ holds, thus v_1^* is the L-critical candidate of $P[v_i, v_j]$. Otherwise $v_1^* \in V_1$ and $v_2^* \in V_2$, then v_1^* achieves $\text{cost}_u^L(W, C, P_1)$ and v_2^* also achieves $\text{cost}_u^L(W, C, P_2)$, respectively. We then compare these two costs and choose one whose cost is larger. \blacktriangleleft

5.1 Basic algorithms

Let us first design an algorithm for **Task 1**, referring to Fig. 2, which shows a part of the CC-tree \mathcal{T} . Here is an informal description of the algorithm.

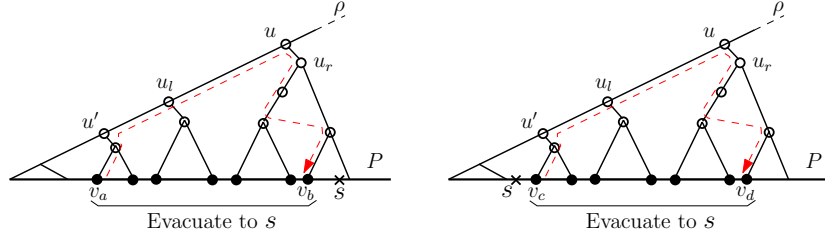


Figure 2 *Left*: Looking for rightmost v_b such that the evacuees from $P[v_a, v_b]$ can evacuate to sink s within time t ; *Right*: Looking for rightmost v_d whose evacuees can evacuate to sink s within time t .

► Algorithm 1. Isolate-subpath(t, v_a)

1. Start at leaf $u = v_a$ of \mathcal{T} ,² and move up towards its root ρ . At each node u visited, do $L\text{-test}(P[v_a, v_R(u)], t)$, i.e., compute the cost of the L-critical vertex of subpath $P[v_a, v_R(u)]$ w.r.t. $v_R(u)^+$, say $\Theta_L(v_a, v_R(u))$. If $L\text{-test}(P[v_a, v_R(u)], t)$ returns "yes", i.e., $\Theta_L(v_a, v_R(u)) \leq t$, then move to its parent node p and set $u = p$. If $L\text{-test}(P[v_a, v_R(u)], t)$ returns "no", then move to the right child node u_r and set $u = u_r$. Start moving down towards a leaf.
2. At each node visited during moving down, do $L\text{-test}(P[v_a, v_R(u)], t)$. If "yes", then move to its parent's right child node p_r and set $u = p_r$. If "no", then move to the left child

² See the left figure in Fig. 2.

node u_l and set $u = u_l$. If u comes to a leaf, say v_b , locate the 1-sink $s \in (v_b, v_{b+1}]$ to the left as much as possible.

3. Start from the vertex v_c that lies immediately to the right of s .³ Performing an up-down search similar to above 1 and 2 so that $R\text{-test}(P[v_c, v_R(u)], s, t)$ is done at each visited node u , determine the rightmost vertex v_d whose evacuees can reach sink s within time t .

► **Lemma 4.** *Assume that the CC-tree, \mathcal{T} , is available. Then $\text{Isolate-subpath}(t, v_a)$ runs in $O(\log^3 n)$ time.*

Proof. At each node u , $\text{Isolate-subpath}(t, v_a)$ carries out $L\text{-test}(P[v_c, v_R(u)], t)$ or $R\text{-test}(P[v_c, v_R(u)], s, t)$. Each of them needs to find the critical vertex by comparing $O(\log n)$ critical candidates. A critical candidate can be computed in $O(\log n)$ time by Lemma 3. Since $\text{Isolate-subpath}(t, v_a)$ visits $O(\log n)$ nodes, the total time is $O(\log^3 n)$. ◀

► **Algorithm 2.** $\text{Find-1sink}(v_i, v_j)$

1. Let u be the node where the two paths $\pi(v_i, \rho)$ and $\pi(v_j, \rho)$ meet.
2. If the L-critical vertex of $P[v_i, v_R(u_l)]$ and the R-critical vertex of $P[v_L(u_r), v_j]$ have the same cost⁴ at some point x on the edge $(v_R(u_l), v_L(u_r))$, then return x as the 1-sink.
3. If the L-critical vertex has a higher (resp. lower) cost than the R-critical vertex at every point on edge $(v_R(u_l), v_L(u_r))$, then let $u = u_l$ (resp. $u = u_r$) and repeat Step 2, using the new u_l and u_r .

Using the arguments similar to those in the proof of Lemma 4, we can prove the following lemma.

► **Lemma 5.** *Assume that the CC-tree, \mathcal{T} , is available. Then $\text{Find-1sink}(v_i, v_j)$ finds the 1-sink on a given subpath $P[v_i, v_j]$ in $O(\log^3 n)$ time.*

5.2 (t, k) -feasibility test

Our approach is to find the maximal subpath from the left end of P for which a 1-sink can achieve completion time t .

► **Lemma 6.** *Given a dynamic path network with n vertices, assume that its CC-tree, \mathcal{T} , is available. Then we can test its (t, k) -feasibility in $O(k \log^3 n)$ time.*

Proof. Starting at the leftmost vertex v_1 of P , invoke $\text{Isolate-subpath}(t)$, which isolates the first subpath in $O(\log^3 n)$ time, and remove it from P . We repeat this at most $k - 1$ more times on the remaining subpath, spending $O(k \log^3 n)$ time. The problem instance is (t, k) -feasible if and only if the rightmost vertex v_n belongs to the last isolated subpath. ◀

5.3 Uniform edge capacity case

The problem is much simplified if the edges have the same capacity. In particular, we can compute the critical vertex of a subpath resulting from concatenating two subpaths in constant time. At each node u of \mathcal{T} bottom up, we compute and record the L- and R-critical vertices of $P[v_L(u), v_R(u)]$ w.r.t. $v_R(u)^+$ and their costs, based on the following lemma.

³ See the right figure in Fig. 2.

⁴ These costs can be computed in $O(\log^3 n)$ time as we saw above.

► **Lemma 7.** [10] *For a node u of CC-tree \mathcal{T} , let $v_L(u_l) = v_h$, $v_R(u_l) = v_j$, $v_L(u_r) = v_{j+1}$, and $v_R(u_r) = v_l$, and assume that the critical vertices, $c_L^{[h,j]}$, $c_R^{[h,j]}$, $c_L^{[j+1,l]}$, and $c_R^{[j+1,l]}$ have already been computed.*

- (a) *The L-critical vertex $c_L^{[h,l]}$ is either $c_L^{[h,j]}$ or $c_L^{[j+1,l]}$.*
- (b) *The R-critical vertex $c_R^{[h,l]}$ is either $c_R^{[h,j]}$ or $c_R^{[j+1,l]}$.*

For example, to the cost of the L-critical cluster $C_L^{[h,j]}$ of $P[v_h, v_j]$, we add the distance cost $d(v_j, v_l)\tau$, and to the cost of the L-critical cluster $C_L^{[j+1,l]}$ of $P[v_{j+1}, v_l]$ we just add $W[v_h, v_j]/c$ to compute its new cost. The L-critical cluster of the combined path $P[v_h, v_l]$ is whichever is larger.

► **Lemma 8.** *Given a dynamic path network with n vertices and uniform edge capacities, we can construct search tree \mathcal{T} in $O(n)$ time and $O(n)$ space.*

Thanks to Lemma 8, Algorithm **Isolate-subpath**(t, v_a) runs in $O(\log n)$ time. We thus have

► **Lemma 9.** *Given a dynamic path network with n vertices and uniform edge capacities, assume that its search tree, \mathcal{T} , is available. Then we can test its (t, k) -feasibility in $O(k \log n)$ time.*

Algorithm **Find-1sink**(v_i, v_j) also runs in $O(\log n)$ time, which implies

► **Lemma 10.** *Given a dynamic path network with n vertices and uniform edge capacities, assume that its search tree, \mathcal{T} , is available. Then we can find the 1-sink on subpath $P[v_i, v_j]$ in $O(\log n)$ time.*

6 Optimization

► **Lemma 11.** [1] *If (t, k) -feasibility can be tested in $T(t, k)$ time, then the k -sink can be found in $O(T(t, k) \cdot k \log n)$ time, excluding the preprocessing time.*

By Lemma 2 it takes $O(n \log n)$ time to construct \mathcal{T} with weight and capacity data, and $T(t, k) = O(k \log^3 n)$ by Lemma 6. We thus have

► **Theorem 12.** *Given a dynamic path network with n vertices, we can find an optimal k -sink in $O(n \log n + k^2 \log^4 n)$ time.*

Based on Lemmas 8 and 9, Megiddo's theorem in [13] implies (it also follows Lemma 11)

► **Theorem 13.** *Given a dynamic path network with n vertices and uniform edge capacities, we can find an optimal k -sink in $O(n + k^2 \log^2 n)$ time.*

6.1 Sorted matrix approach

Let $OPT(l, r)$ denote the evacuation time for the optimal 1-sink on subpath $P[v_l, v_r]$. Define an $n \times n$ matrix A whose entry (i, j) entry is given by

$$A[i, j] = \begin{cases} OPT(n - i + 1, j) & \text{if } n - i + 1 \leq j \\ 0 & \text{otherwise.} \end{cases} \quad (23)$$

It is clear that matrix A includes $OPT(l, r)$ for every pair of integers l and r such that $1 \leq l \leq r \leq n$. There exists a pair of integers l and r such that $OPT(l, r)$ is the evacuation

time for the optimal k -sink on the whole path. Then k -sink location problem can be written as: “Find the smallest $A[i, j]$ such that the given problem instance is $A[i, j]$ -feasible.”

A matrix is called a *sorted matrix* if each row and column of it is sorted in the nondecreasing order. In [4, 5], Frederickson et al. show how to search for such a minimum in a sorted matrix. The following lemma is implicit in their papers.

► **Lemma 14.** *Suppose that $A[i, j]$ can be computed in $f(n)$ time, and feasibility can be tested in $g(n)$ time. Then we can solve the k -sink problem in $O(nf(n) + g(n) \log n)$ time.*

We have $f(n) = O(\log^3 n)$ by Lemma 5, and $g(n)$ can be $O(n \log^2 n)$ by scanning path P from left to right. Lemma 14 thus implies

► **Theorem 15.** *Given a dynamic path network with n vertices and general edge capacities, we can find an optimal k -sink in $O(n \log^3 n)$ time.*

In the uniform capacity case, we can show that $f(n) = O(\log n)$, and $g(n)$ can be $O(n)$ by scanning the path from left to right. Lemma 14 thus implies

► **Theorem 16.** *Given a dynamic path network with n vertices and uniform edge capacities, we can find the k -sink in $O(n \log n)$ time.*

7 Conclusion and discussion

We have shown that on dynamic path networks with n vertices, the k -sink can be found in $O(\min\{n + k^2 \log^4 n, n \log^3 n\})$ time, which is sub-quadratic. If the edges have the same capacity, we can solve the k -sink problem in $O(\min\{n + k^2 \log^2 n, n \log n\})$ time. These results improve upon the previously best algorithms [1, 10] for all values of k .

References

- 1 Guru Prakash Arumugam, John Augustine, Mordecai J. Golin, Yuya Higashikawa, Naoki Katoh, and Prashanth Srikanthan. Optimal evacuation flows on dynamic paths with general edge capacities. *arXiv:1606.07208v1*, 2016.
- 2 Siu-Wing Cheng, Yuya Higashikawa, Naoki Katoh, Guanqun Ni, Bing Su, and Yinfeng Xu. Minimax regret 1-sink location problem in dynamic path networks. In *Proc. Annual Conf. on Theory and Applications of Models of Computation (T-H.H. Chan, L.C. Lau, and L. Trevisan, Eds.), Springer-Verlag*, volume LNCS 7876, pages 121–132, 2013.
- 3 Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications, Third Edition*. Springer Verlag, 2008.
- 4 G.N. Frederickson. Optimal algorithms for tree partitioning. In *Proc. 2nd ACM-SIAM Symp. Discrete Algorithms*, pages 168–177, 1991.
- 5 G.N. Frederickson and D.B. Johnson. Finding k th paths and p -centers by generating and searching good data structures. *J. Algorithms*, 4:61–80, 1983.
- 6 Ronald L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information processing letters*, 1(4):132–133, 1972.
- 7 H.W. Hamacher and S.A. Tjandra. Mathematical modelling of evacuation problems: a state of the art. in: *Pedestrian and Evacuation Dynamics, Springer Verlag*,, pages 227–266, 2002.
- 8 Yuya Higashikawa. *Studies on the space exploration and the sink location under incomplete information towards applications to evacuation planning*. PhD thesis, Kyoto University, Japan, 2014.

- 9 Yuya Higashikawa, Mordecai J. Golin, and Naoki Katoh. Minimax regret sink location problem in dynamic tree networks with uniform capacity. *J. of Graph Algorithms and Applications*, 18.4:539–555, 2014.
- 10 Yuya Higashikawa, Mordecai J. Golin, and Naoki Katoh. Multiple sink location problems in dynamic path networks. *Theoretical Computer Science*, 607:2–15, 2015.
- 11 S. Mamada, K. Makino, and S. Fujishige. Optimal sink location problem for dynamic flows in a tree network. *IEICE Trans. Fundamentals*, E85-A:1020–1025, 2002.
- 12 Satoko Mamada, Takeaki Uno, Kazuhisa Makino, and Satoru Fujishige. An $O(n \log^2 n)$ algorithm for a sink location problem in dynamic tree networks. *Discrete Applied Mathematics*, 154:2387–2401, 2006.
- 13 N. Megiddo. Combinatorial optimization with rational objective functions. *Math. Oper. Res.*, 4:414–424, 1979.
- 14 Franco P Preparata and Michael Shamos. *Computational geometry: an introduction*. Springer Science & Business Media, 2012.